CS 581, Winter 2024 Homework 1 (Haskell)

Important Reminders!

1. Upload your solution as a Haskell file (ending in `.hs`) to Canvas.
2. **Only submit files that compile without errors!** (Put all non-working parts in comments.)
3. You must do all homework assignments by yourself, without the help of others. Also, you must not use services such as Chegg or Course Hero. If you need help, simply ask on Canvas, and we will help!
4. The homework is graded leniently, and we reward serious efforts, even when you can't get a correct solution.

Exercise 1. Programming with Lists

Multisets, or bags, can be represented as list of pairs (x, n) where n indicates the number of occurrences of x in the multiset.

```haskell
type Bag a = [(a,Int)]
```

For the following exercises you can assume the following properties of the bag representation. *But note:* Your function definitions have to maintain these properties for any multiset they produce!

(1) Each element x occurs in at most one pair in the list.
(2) Each element that occurs in a pair has a positive counter.

As an example consider the multiset $\{2, 3, 3, 5, 7, 7, 7, 8\}$, which has the following representation (among others).

```
[(5,1),(7,3),(2,1),(3,2),(8,1)]
```

Note that the order of elements is not fixed. In particular, we cannot assume that the elements are sorted. Thus, the above list representation is just one example of several possible.

(a) Define the function `ins` that inserts an element into a multiset.

```haskell
ins :: Eq a => a -> Bag a -> Bag a
```

(*Note:* The class constraint "`Eq a =>`" restricts the element type `a` to those types that allow the comparison of elements for equality with `==`.)

(b) Define the function `del` that removes a single element from a multiset. Note that deleting 3 from $\{2, 3, 3, 4\}$ yields $\{2, 3, 4\}$ whereas deleting 3 from $\{2, 3, 4\}$ yields $\{2, 4\}$.

```haskell
del :: Eq a => a -> Bag a -> Bag a
```

(c) Define a function `bag` that takes a list of values and produces a multiset representation.

```haskell
bag :: Eq a => [a] -> Bag a
```

For example, with `xs = [7,3,8,7,3,2,7,5]` we get the following result.

```
> bag xs
[(5,1),(7,3),(2,1),(3,2),(8,1)]
```

(*Note:* It's a good idea to use of the function `ins` defined earlier.)

(d) Define a function `subbag` that determines whether or not its first argument bag is contained in the second.

```
subbag :: Eq a => Bag a -> Bag a -> Bool
```

Note that a bag b is contained in a bag b' if every element that occurs n times in b occurs also at least n times in b'.

(e) Define a function `isSet` that tests whether a bag is actually a set, which is the case when each element occurs only once.

```
isSet :: Eq a => Bag a -> Bool
```

(f) Define a function `size` that computes the number of elements contained in a bag.

```
size :: Bag a -> Int
```

Exercise 2. Matrix Operations

Suppose we implement vectors and matrices in Haskell by lists. This means we use the following types (`Row` and `Column` are exactly the same types, they are used to better document the functions).

```
type Row    = [Int]
type Column = [Int]
type Matrix = [Row]
```

The goal is to find Haskell function definitions for matrix operations based on this representation. (If you are not familiar with the definitions of matrix operations, you might want to check: `https://en.wikipedia.org/wiki/Matrix_(mathematics)`.) You may assume in your definitions that all the rows in a matrix have the same length.

(a) Give a function definition for vector addition, that is, define the Haskell function `vAdd` of type:

```
vAdd :: Row -> Row -> Row
```

After having defined the function using recursion, take a look at the function `zipWith`, which takes a function `f` and two lists `l1` and `l2` and and creates a new list whose elements are obtained by applying `f` to the elements `l1` and `l2` in parallel (like they were vectors). For example:

```
> zipWith (*) [1,2,3] [4,5,6]
[4,10,18]
```

When applied to two lists of different lengths, `zipWith` stops after the elements of the shorter list are exhausted. Give a definition of `vAdd` that uses `zipWith`.

(b) Define a function `mAdd` for adding two matrices. (Remember the function `zipWith`.)

```
mAdd :: Matrix -> Matrix -> Matrix
```

(c) Define the function `iProd` for computing the inner product (also called dot product).

```
iProd :: Row -> Column -> Int
```

(d) Give a function definition `oProd` for computing the outer product:

```
oProd :: Column -> Row -> Matrix
```

(e) Define a function that computes the size of a matrix as a pair (n,m) where n gives the number of rows and m gives the number of columns.

```
mSize :: Matrix -> (Int,Int)
```

Exercise 3. Higher-Order Functions

(a) Define a higher-order function `applyAll` that takes a list of functions `fs` and a value `x`, applies all functions in the list `fs` to `x`, and returns the list of all results. Here are two possible application for `applyAll`.

```
> applyAll [even,(<9)] 7
[False,True]

> applyAll [(^2),succ] 7
[49,8]
```

The two most obvious implementations are using recursion or list comprehensions. Any such solution is perfectly fine. However, a more elegant solution can be obtained using the function `map`.

(b) We can view a function of type `a -> Bool` as representing basic properties (of values of type `a`). We define the type `Property` for representing composite properties as a collection of such functions.

```
type Property a = [a -> Bool]
```

As an example, consider the the property of a number being a digit. It can be represented as a value of type `Property` as follows.

```
digit :: Property Int
digit = [(>=0),(<10)]
```

Using the function `applyAll` from part (a) and the predefined function `and :: [Bool] -> Bool` that checks whether all values in a list of booleans are `True`, define a function `satisfies` that checks whether a value satisfies a composite property.

```
satisfies :: Property a -> a -> Bool
```

(c) Define a function `power` that applies a function repeatedly (that is, exactly n times) to a value. In other words, for a function f and a value x, `power` computes $f^n(x)$.

```
power :: (a -> a) -> Int -> a -> a
```

Here are two example usages of `power`.

```
> power tail 3 [1..10]
[4,5,6,7,8,9,10]

> power ("Ho"++) 5 ""
"HoHoHoHoHo"
```

(d) Define the following two functions for adding and multiplying integers using the function `power` from part (c). (*Note*: For the definition of `plus` you should use `succ` as the argument function for `power`.)

```
plus :: Int -> Int -> Int
times :: Int -> Int -> Int
```